# Generation of SDN policies for protecting Android environments based on automata learning

Nicolas Schnepf, Rémi Badonnel, Abdelkader Lahmadi, Stephan Merz

# Generation of SDN Policies for Protecting Android Environments based on Automata Learning

Nicolas Schnepf, Rémi Badonnel, Abdelkader Lahmadi, Stephan Merz
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
Email: {nicolas.schnepf, remi.badonnel, abdelkader.lahmadi, stephan.merz}@inria.fr

***Abstract*—Software-defined networking offers new opportunities for protecting end users and their applications. In that context, dedicated chains can be built to combine different security functions, such as firewalls, intrusion detection systems and services for preventing data leakage. To configure these security chains, it is important to have an adequate model of the patterns that end user applications exhibit when accessing the network. We propose an automated strategy for learning the networking behavior of end applications using algorithms for generating finite state models. These models can be exploited for inferring SDN policies ensuring that applications respect the observed behavior: such policies can be formally verified and deployed on SDN infrastructures in a dynamic and flexible manner. Our solution is prototypically implemented as a collection of Python scripts that extend our Synaptic verification package. The performance of our strategy is evaluated through extensive experimentations and is compared to the Synoptic and Invarimint automata learning algorithms.**

## I. Introduction

The increasing number of connected devices, such as smartphones, smart objects and sensors, is an important factor of Internet growth and dynamics. It also poses new challenges in terms of security management. These devices with restricted capacities generate a larger attack surface. A typical example is the recent attack against the DynDNS service, which exploited a botnet of infected smart objects to perform flooding. Malicious applications targeting these devices are also increasing massively every year. Preventive security methods that consist in analyzing the applications before their publication on markets are of limited effectiveness. For instance, analyses estimate that more than 3 million malwares were published on the official Google application market [1], [2]. Security mechanisms must be adjusted to threats dynamically. They are also constrained by the resources of devices that are often limited in terms of memory and CPU, making the local deployment of protective mechanisms more challenging.

In the meantime, the development of software-defined networking (SDN) provides new perspectives in terms of security, through an extended programmability of networks [3], [4]. The SDN architecture relies on the decoupling of the data and control planes of networks, the first consisting in programmable switches deployed in the network, and the second consisting in a central component, the controller, responsible for dynamically configuring the switches in response to network events. The interface between these planes is implemented by a dedicated protocol, such as OpenFlow, that can then serve as a support for deploying security rules from the controller to the switches. Programming the controller can benefit from high-level languages, such as Pyretic [5], part of the Frenetic family of languages [6]. These languages allow the dynamics of the controller and the forwarding rules of the switches to be described as a Python program, before compiling them into low-level OpenFlow rules.

This programmability brings flexibility to the deployment and adjustment of security mechanisms in the network infrastructures. In this paper we propose an approach based on automata learning to capture the networking behavior of user applications; such models can be used for generating SDN policies in order to protect Android environments. More precisely, we build finite-state models that characterize the behavior of user applications through the analysis of flow-based interaction traces. We aim to infer SDN policies from these models in order to protect end users. Security policies can be expressed using Pyretic rules, for which formal verification methods are available that can help validating network policies. The verification of the control plane can be performed by model checking using the Kinetic extension [7]. In addition, the data plane associated to Pyretic rules can be verified using our Synaptic checker described in [8].

Our main contributions are: (i) designing an automated technique for learning the behavior of Android applications, (ii) developing practical algorithms for building the automata, (iii) prototyping our method based on Python scripts, and (iv) evaluating its performance through extensive experiments, with a comparison to other automata learning algorithms, namely Synoptic [9] and Invarimint [10].

The remainder of this paper is organized as follows. Section II gives an overview of existing related work. Section III describes the envisaged architecture for generating SDN policies and our algorithms for building finite state models. Section IV compares performance results obtained with different automata learning methods. Section V concludes and points out future research perspectives.

## II. Related work

The widespread presence of malware applications on the Google market store has spurred research on the validation of Android applications. Asavoae et al. describe techniques for detecting collusion of Android malware applications based on their permissions [11], but do not address the detection of attacks from a networking perspective. Similarly, Zhang et al. rely on the Android permissions for automatically

generating security centric descriptions of applications [12], but also do not consider the validation of networking aspects. Mariconti et al. detect malware by building Markov models of the applications, but this approach requires downloading and executing the byte code of the application [13]. Kim et al. mine the network protocols of an application from its executable, but the deployment of this approach is intractable in SDN networks [14]. Finally, Xia et al. audit Android applications at runtime [15], and Ren et al. propose an approach for discovering personal data in the network traffic of an application [16]; however, these approaches do not include automatic adjustments or the verification of network policies.

In our previous paper [8] we introduce the Synaptic checker for the verification of both control and data planes of a software-defined network policy. This checker relies on the Pyretic programming language [5], part of the Frenetic family of languages for programming SDN controllers [6]: Pyretic is implemented as a domain-specific language embedded in Python for the specification of chains of rules at an abstract level, from which low-level OpenFlow rules can be compiled. Pyretic is complemented by an extension, called Kinetic, for describing control plane policies; Kinetic also offers formal verification techniques based on model checking [7]. Synaptic extends this formalism for verifying the correctness of both the control and data planes of Pyretic policies, before their deployment in the network.

To automate the protection of Android applications, we suggest here to complement our Synaptic checker by automata learning methods for obtaining a representation of the network behavior of Android applications. Automata learning methods can be classified into several categories: purely algorithmic, purely statistical or a mix of both. Among the purely algorithmic methods, the reference $k$-tails algorithm [17] can mine exact, but potentially complicated, Markov models of applications. This algorithm underlies the Synoptic[1] tool [9] that can learn Markov models of processes. The Invarimint tool [18] is also based on $k$-tails but produces simpler, so-called declarative automata, without probabilities on the edges. In the field of statistical automata learning algorithms, there is the approach proposed by Carrasco [19] for inferring grammar of languages by stochastic methods used in [20] for learning inexact automata from logs of behaviors. Das and Mozer [21] suggest relying on neural nets for learning models of applications; this approach requires a learning session before producing the model of a system while exact approaches can be directly applied on logs without training phase.

## III. Automata Learning Strategy for Supporting Android Application Security

We propose in this paper an approach for learning interaction models of applications for protecting Android environments. More precisely, this approach consists in automatically profiling applications based on automata learning methods applied to network logs. We mainly focus on the network traces characterizing the behavior of applications, in contrast to Android permissions that focus on the services that applications can access directly on the smart device. One question to consider is whether application models should be built from packets or flows: a packet is the basic unit of data exchanged by applications during a communication whereas a flow is an aggregated and unidirectional collection of packets that can be used instead of packets for representing the network behavior of applications.

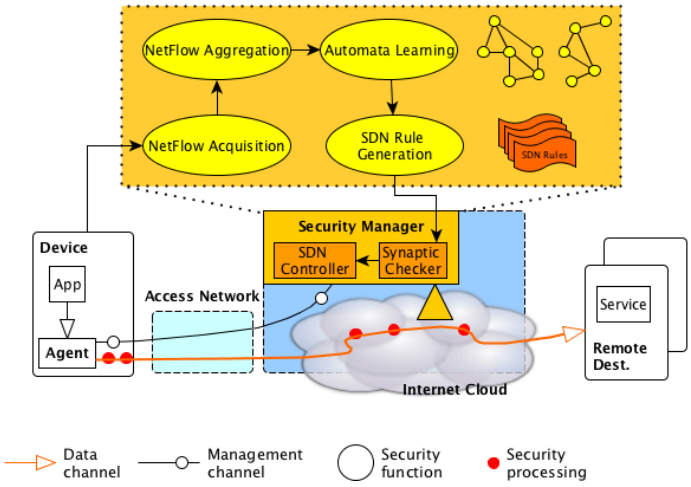


Figure 1. Overview of the proposed architecture for securing applications.

We decided to build our models from flow traces, following Sperotto [22] who argues that using flows instead of packets presents several advantages from a security perspective. First of all, analyzing flows is faster than analyzing packets because we are interested in collections of packets sharing the same IP addresses and port numbers. A second advantage of this approach is higher confidentiality for end users: whereas an analysis at the level of packets can observe the data exchanged during a communication, the latter is hidden in the analysis of flows: one can only observe aggregate information such as the numbers of packets and of bytes but not what is transmitted over the network. We are mainly interested in the remote servers the application is communicating with, focusing on identifying communications that may present a risk for the safety of a smart environment.

The communication pattern of an application can conveniently be represented as a finite state automaton; in particular we consider hidden Markov models of applications because this representation captures interesting properties in terms of network security [22]. In addition to probabilistic aspects we are also interested in having quantitative information about the numbers of bytes and of packets sent or received by applications during their communications in order to capture abnormal behaviors and to adjust the chains accordingly when observing such events at the network level.

### A. Underlying architecture

The overall architecture that supports our technique for protecting devices and applications in an SDN environment

[1]Synoptic is an automata miner, while Synaptic is a checker.

is depicted in Figure 1. It relies on a security manager that is responsible for the orchestration of security chains on top of an SDN controller. Our approach is organized into four major phases: netflow acquisition, netflow aggregation and automata learning. The acquisition probe is deployed directly on the smart device for collecting application flows. Using a dedicated protocol, these traces are sent to the security manager. After an aggregation phase (realized by an abstraction module), a process miner is applied during the automata learning phase for building a finite state machine that captures the communication behavior of the application.

### B. NetFlow acquisition

The first phase relates to the acquisition of flow traces of applications that are collected on the end device. In the context of protecting Android devices, this acquisition is supported by Flowoid [23], a probe developed in our research group. Deployed on Android devices, Flowoid collects traces of the network behavior of running applications and computes several statistics: the transmission of these traces to the security manager is based on NetFlow, a protocol specially designed for the acquisition of network flows over the network. In terms of information, Flowoid collects for each flow the IP addresses of the source and of the destination and the corresponding port numbers: it also provides information on the network protocol (TCP or UDP), a timestamp labeling the communication, its length, and the numbers of bytes and of packets transmitted. In order to build a model representing the behavior of applications, the fields that characterize the communications of an application must be identified: actually the IP source and its corresponding port number are not informative because they are bound to the smart device. The information about the network protocol and the numbers of bytes and packets are interesting but not sufficient for building a representative model of the communications of an application. The most relevant fields are the IP destination that defines the remote server that provides the service and the corresponding port number that defines this service. We will therefore initially consider only these fields for building the transitions of our model. The other informations are aggregated in order to infer properties of the traffic, as we describe next.

### C. NetFlow aggregation

We rely on flows collected by Flowoid for inferring our application models: however, we need to define a way of aggregating these flows, in order to infer interesting properties on the traffic. If we directly use the IP addresses and port numbers contained in the logs, we obtain very precise application models. The problem of this approach is that whenever the IP address of a service changes, our models do not hold anymore. We therefore need to consider the traffic at a higher level of abstraction. An idea for solving this problem is to consider the type of service the application is communicating with, for instance google or amazon. In other words, we are interested in knowing the owner of the IP addresses contained in a log. This information can be collected by using `whois` requests: this command returns information about the organization owning the IP address in the field *orgname* but it can also return a more precise information, viz. the name of the network in the field *netname*. When `whois` returns both a netname and an orgname we have to decide which information to use for building the automaton: using only netnames could again result in a too precise automaton containing traces of networks appearing only once; using only orgnames could result in automata integrating many flows aggregated under very general services such as RIPE Network Coordination Centre. As a compromise between these two extremes, we decided to set a threshold $n$: if a netname appears more than $n$ times in the overall log it is considered as significant and it is integrated into the model of the application; otherwise we integrate the orgname. We thereby limit the influence of netnames appearing only once or twice in the input log by setting a high value of $n$; conversely, we can avoid a too strong presence of generic orgnames by setting a lower value of $n$. We complement this strategy for aggregating IP addresses by another one for handling port numbers: although most Android applications only use HTTP or HTTPS ports and do not require such an aggregation strategy, some applications present a high disparity of port numbers that should be aggregated. In that case we can use the most frequent prefixes of port numbers in order to aggregate flows: this approach allows us to group together flows that have the same port number; moreover, when an application frequently communicates on a given port range we can aggregate these flows in order to get a more compact representation of the traffic of applications. The idea is again to only consider prefixes that appear more often than a certain threshold in order to limit the influence of port numbers appearing only once or twice in the flow.

### D. Automata learning

The next phase consists in automata learning. The objective is to learn a Markovian model capturing the network behavior of an application. This automaton is implemented by two tables: $States$ associates flows to their number of instances, while $Transitions$ associates pairs of flows to their probability of succession. The arguments of our algorithm (see Algorithm 1) are the list noted $Flows$ of extended flows computed by our aggregation module and the size $N$ of this list; the local variables that we use are $flow$ containing the orgname and the port number associated to a flow and $transition$ containing two consecutive flows. Each state of the automaton is a class of flows defined by the name of the destination organization and the corresponding port number. Moreover, states can be enriched by auxiliary information in order to represent more details about the corresponding traffic. Concretely, we compute the following standard network information for each state of our automata:

- the maximum numbers of bytes and of packets sent during a flow;
- the average number of bytes and of packets sent during a flow;
- the length of the longest flow;

**Algorithm 1** Automaton learning algorithm.

$States := \emptyset$
$Transitions := \emptyset$
$Flows$ :=List of flows.
▷ Initialize the set of states
$flow := Flows[0]$
$States[flow] := 1$
▷ Count occurrences of states and transitions
**for** $i \in 1..N$ **do**
  $transition := (flow, Flows[i])$
  $flow := Flows[i]$
  **if** $flow \in States$ **then**
    $States[flow] \mathrel{+}= 1$
  **else**
    $States[flow] := 1$
  **end if**
  **if** $transition \in Transitions$ **then**
    $Transitions[transition] \mathrel{+}= 1$
  **else**
    $Transitions[transition] := 1$
  **end if**
**end for**
▷ Compute the probabilities of transitions
**for** $transition \in Transitions$ **do**
  $Transitions[transition] :=$
  $Transitions[transition]/States[transition_0]$
**end for**

| Applications | Number of NetFlow logs | Number of IP/ports |
|---|---|---|
| disneyland | 282 | 5 |
| dropbox | 1000 | 17 |
| faceswitch | 151 | 30 |
| lequipe | 1000 | 208 |
| meteo | 1000 | 89 |
| ninegag | 1000 | 124 |
| pokemongo | 275 | 24 |
| ratp | 779 | 3 |
| skype | 1000 | 442 |
| viber | 1000 | 176 |

Figure 2. Set of applications used during our experiments.

- the average length of the set of flows;
- the rates of TCP/UDP traffic.

As an example, we consider the logs of 275 flows generated by the pokemongo application: these traces contain communications of the application with 24 different hosts on the ports 80 and 443. Our aggregation strategy groups the 24 IP addresses of these logs under 7 different orgnames: based on this information we run our automata learning algorithm and obtain the automaton of Figure 3. This automaton contains 10 states and 33 transitions; to give a comparison, the automata mined directly from the IP addresses contains 26 states and 77 transitions. We also applied Invarimint and Synoptic to this example: Invarimint produces an automaton with 12 states and 32 transitions without probabilities and Synoptic produces an automaton with 28 states and 65 transitions.

## IV. Performance Evaluation

We evaluated the performance of our prototype through several experiments. In particular, we wanted to compare the performances obtained with different methods of automata learning. The experimental setup was based on a MacBook Air laptop computer with an Intel Core i5 (1.7 GHz) processor and 4Gb of RAM. We considered the three following process/automata miners: Synoptic, its declarative version implemented in Invarimint, and finally our own generation method described in the previous section. For these experiments we used a set of 10 log files, described in Figure 2. From these log files we built the automata with the three methods that we selected, noting that we were not able to get the automata for the applications lequipe, skype and viber with Synoptic because of excessive memory consumption.

In order to compare the automata that we obtained we defined the following evaluation criteria:

- the simplicity of the resulting automata, expressed in terms of numbers of both states and transitions;
- the precision of the model, expressed as the number of flows of other applications rejected by the automata.

### A. Simplicity of automata

We only considered exact generation methods in this evaluation: the automata produced by such algorithms are guaranteed to accept all the logs used for learning. However, such automata can be quite complicated to read and to interpret, and their size has an influence on the number of SDN rules that are generated. We therefore chose their simplicity as an evaluation criterion, measured as the numbers of states and of transitions of an automaton. Because these metrics vary with both the method of generation and with the level of abstraction of the input logs, we compared the different methods by applying them on logs aggregated with different values of the threshold parameter $n$. The results of these experiments are presented in Figure 4.

Without aggregating IP addresses our approach produces automata with in average 105.2 states and 322.8 transitions for this data set. The automata produced by Invarimint and by our approach are of similar complexity on average, Invarimint produces automata that have two more states and one transition less than our approach. In contrast, Synoptic produces automata with in average 11.2 states and 29 transitions more than our approach. Indeed, the automata generated by Synoptic are always the most complex ones. Considering the performances of the aggregation strategy, the best improvement is obtained when replacing the IP addresses by their corresponding orgnames; concerning the influence of the value of $n$, the strongest impact is observed when aggregating the less frequent netnames, meaning that limiting the influence of values with low frequency of appearance is the most important factor for choosing the threshold value $n$. For completing these results we also compared the automata sizes for our approach and for Invarimint when applying the port abstraction strategy on the logs of the three most

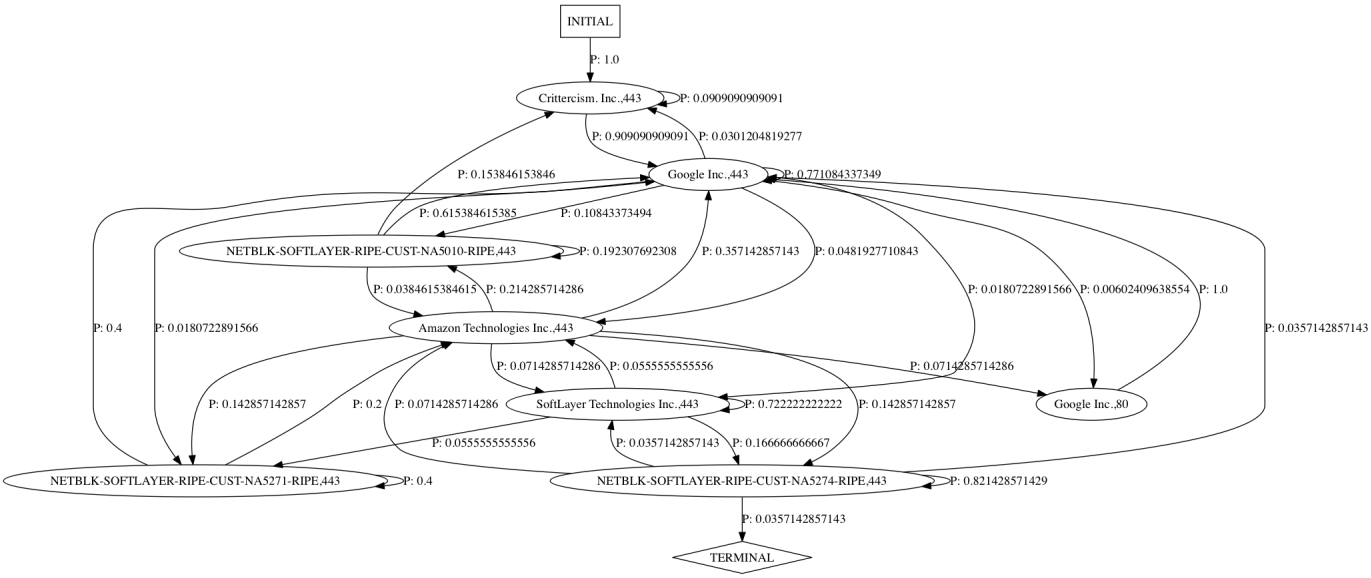


Figure 3. Automaton describing the behavior of the pokemongo application.

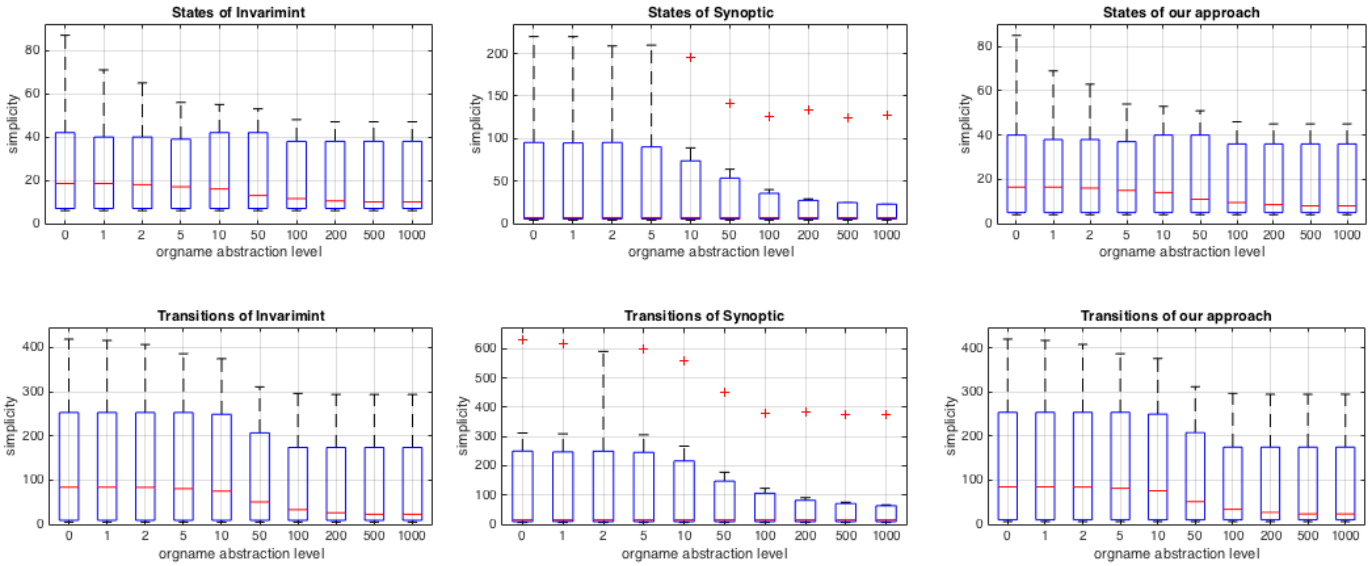


Figure 4. Simplicity of automata generated with the considered methods.

complex applications: lequipe, skype and viber. During these experiments we considered the influence of the threshold on port numbers varying from 0 to 10. These results are described in Figure 5.

The high number of states and of transitions of these automata is caused by the complexity of the applications. The complexity of the automata decreases most significantly for threshold values between 0 and 3. This can be explained by the peer-to-peer profile of the considered applications that causes many port numbers to appear only one or two times. Port numbers appearing more than 4 or 5 times can be considered as being representative of the behavior of the application, nevertheless, for logs where such values would constitute an important part of the traffic it could be interesting to consider higher threshold values.

### B. *Precision of automata*

The next criterion that we considered is the precision of the automata: given an application, we define the precision of its automaton as the percentage of logs of other applications that it rejects, in other words as the rate of foreign traffic rejected by the automaton. Having a high precision is important for ensuring that the traffic of an application will not be blocked by other automata. On the other hand, if two automata match the same traffic, their respective sets of rules could be joined into a set that is useful for both applications. We evaluated the precision of the automata generated by each method

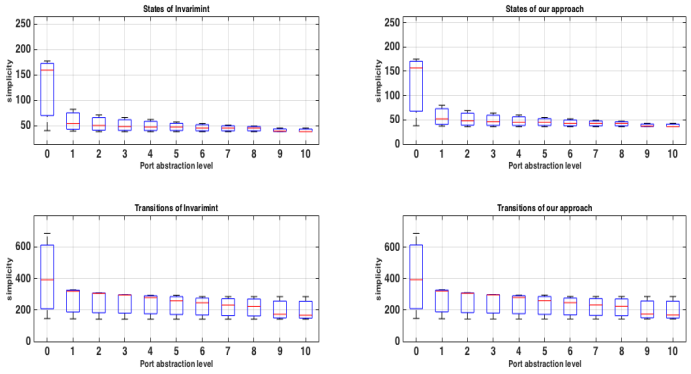


Figure 5. Impact of port abstraction on the sizes of automata.

for the different applications, when using the orgname-based abstraction: these results are depicted in Figure 6.

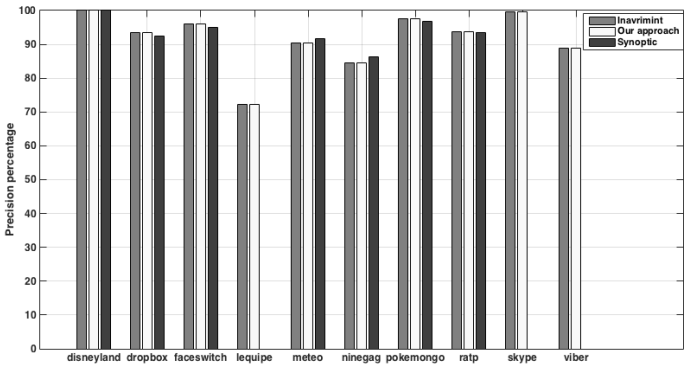


Figure 6. Precision of orgname-based automata.

One can observe that the precision depends more on the type of application that is considered than on the method used for generating the automata. Moreover, for some applications, using the orgnames for computing the automata is sufficient for having a satisfactory precision. However for others, such as lequipe, precision is unsatisfactory. This difference is due to the fact that some applications only contact private services such as disney world wide service whereas others contact many public services such as Amazon or Google. For completing these results we ran another set of experiments with automata generated from the IP addresses of the remote hosts that the application contacts: these results are presented in Figure 7. Again, the precision depends more on the type of application that is considered than on the method used for generating the automata. We can see that the precision of the automata is better without abstracting the logs[2]: this fact confirms our strategy of using the IP addresses for matching the traffic at the network level. However, the strong overlaps observed when using the orgnames for generating the automata could be exploited for identifying sections of application traffic that can be protected by the same types of chains, so the

[2]Observe the different scale of the ordinate axis in Figure 7.

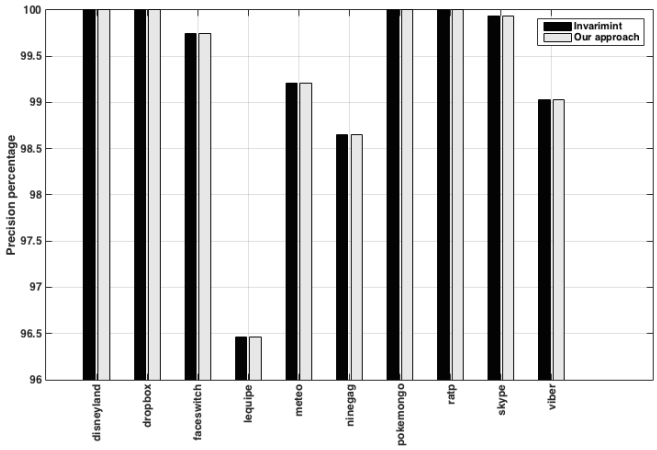


Figure 7. Precision of automata based on IP addresses.

information on the IP addresses would be used for configuring these chains.

## V. Conclusions and Future Work

We propose in this paper a technique for learning the behavior of Android applications, as a starting point for generating SDN policies for protecting them. Our solution collects traces of NetFlow records of their applications, aggregates them, and finally builds finite-state models. We have described the architecture supporting our approach, as well as the interactions among its different components. We have designed and implemented aggregation and automata learning algorithms that allow precise and generic models of applications to be built, with the aim of using these models for configuring chains of security functions specified in the Pyretic language and verified with our Synaptic checker. We have developed a prototype of our solution implementing these algorithms, and evaluated its performances through a series of experiments based on the backend process miners Synoptic and Invarimint, in addition to our own algorithm. The experiments showed the benefits and limits of these methods in terms of simplicity, precision, genericity, and expressivity, while varying the level of aggregation of the input flow traces.

As future perspectives, we work on the generation or selection of chains of security functions, based on the inferred finite-state automata. We are also interested in further optimizing the formal models that we generate, based on the nature of the properties to ensure. Finally, we want to investigate further the integration of this automata learning method into the context of an automated management framework for security chains with our Synaptic checker.

## References

[1] M. La Polla, F. Martinelli, and D. Sgandurra, "A Survey on Security for Mobile Devices," in *IEEE Communications Surveys & Tutorials*, 2012.

[2] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android Security, a Survey of Issues, Malware Penetrations and Defenses," in *IEEE Communications Surveys & Tutorials*, July 2015.

[3] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN, an Intellectual History of Programmable Networks," *SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[4] N. Feamster and H. Kim, "Software-Defined Networks: Improving Network Management with SDN," in *IEEE Communications Magazine*, February 2013.

[5] N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. P. Kata, C. Monsanto, J. Reich, M. Reitblatt, R. Jennifer, C. Schlesinger, A. Story, and D. Walker, "Languages for Software-Defined Networks," in *Software Technology Group*, 2016.

[6] N. Foster, M. J. Freedman, R. Harrison, C. Monsanto, and D. Walker, "Frenetic, a Network Programming Language," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP'11)*, 2011.

[7] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable Dynamic Network Control," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*, 2015.

[8] N. Schnepf, S. Merz, R. Badonnel, and A. Lahmadi, "Automated verification of security chains in software-defined networks with Synaptic," in *Proceedings of the 3rd IEEE Conference on Network Softwarization (NetSoft'17)*, 2017.

[9] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst, "Synoptic: Studying logged behavior with inferred models," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 448–451. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025188

[10] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms," in *IEEE Transactions on Software Engineering*, vol. 41, 2015, pp. 408–428.

[11] I. M. Asavoae, J. Blasco, T. M. Chen, H. K. Kalutarage, I. Muttik, H. N. Nguyen, M. Roggenbach, and S. A. Shaikh, "Towards automated android app collusion detection," in *Proc. 1st Intl. Wsh. Innovations in Mobile Privacy and Security (IMPS 2016)*, ser. CEUR Workshop Proceedings, vol. 1575, London, UK, 2016, pp. 29–37. [Online]. Available: http://ceur-ws.org/Vol-1575

[12] M. Zhang, Y. Duan, Q. Feng, and H. Yin, "Towards automatic generation of security centric descriptions for android apps," in *dings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2015)*, 2015.

[13] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," *CoRR*, vol. abs/1612.04433, 2016. [Online]. Available: http://arxiv.org/abs/1612.04433

[14] J. Kim, H. Choi, H. Namkung, W. Choi, B. Choi, H. Hong, Y. Kim, J. Lee, and D. Han, "Enabling automatic protocol behavior analysis for android applications," in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '16. New York, NY, USA: ACM, 2016, pp. 281–295. [Online]. Available: http://doi.acm.org/10.1145/2999572.2999596

[15] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 899–914. [Online]. Available: http://dx.doi.org/10.1109/SP.2015.60

[16] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. R. Choffnes, "Recon: Revealing and controlling privacy leaks in mobile network traffic," *CoRR*, vol. abs/1507.00255, 2015. [Online]. Available: http://arxiv.org/abs/1507.00255

[17] A. Biermann and J. Feldman, "On the synthesis of finite-state machines from samples of their behavior," in *IEEE Transactions on Computers*, 1972.

[18] M.-Y. Kang, J.-Y. Choi, I. Kang, H. H. Kwak, S. J. Ahn, and M.-K. Shin, *A Verification Method of SDN Firewall Applications*. IEICE Transactions on Communications, 2016.

[19] R. C. Carrasco and J. Oncina, "Learning stochastic regular grammars by means of a state merging method," in *2nd Intl. Coll. Grammatical Inference and Applications (ICGI-94)*, ser. LNCS, R. C. Carrasco and J. Oncina, Eds., vol. 862. Alicante, Spain: Springer, 1994. [Online]. Available: https://doi.org/10.1007/3-540-58473-0_144

[20] J. E. Cook and A. L. Wolf, "Discovering models of software processes from event-based data," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 3, pp. 215–249, Jul. 1998. [Online]. Available: http://doi.acm.org/10.1145/287000.287001

[21] S. Das and M. C. Mozer, "A unified gradient-descent/clustering architecture for finite state machine induction," in *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds. Morgan-Kaufmann, 1994, pp. 19–26.

[22] A. Sperotto, "Flow-based intrusion detection," Ph.D. dissertation, University of Twente, 10 2010, 10.3990/1.9789036530897.

[23] A. Lahmadi, F. Beck, E. Finickel, and O. Festor, "A platform for the analysis and visualization of network flow data of android environments," IFIP/IEEE International Symposium on Integrated Network Management (IM), May 2015, poster. [Online]. Available: https://hal.inria.fr/hal-01242911